DOCUMENT RESUME

ED 305 925 IR 013 794

AUTHOR Perkins, D. N.; And Others
TITLE Conditions of Learning in Novice Programmers. Concept Paper.
INSTITUTION Educational Technology Center, Cambridge, MA.
SPONS AGENCY Office of Educational Research and Improvement (ED), Washington, DC.
REPORT NO ETC-85-13
PUB DATE Apr 85
NOTE 39p.
PUB TYPE Viewpoints (120) -- Reports - Evaluative/Feasibility (142)

EDRS PRICE MF01/PC02 Plus Postage.
DESCRIPTORS Classroom Observation Techniques; *Cognitive Style; Discovery Learning; Elementary Secondary Education; *Learning Processes; *Learning Strategies; Microcomputers; *Problem Solving; *Programing; Psychological Studies; *Skill Development
IDENTIFIERS BASIC Programing Language; Debugging (Computers); LOGO Programing Language

ABSTRACT

Under normal instructional circumstances, some youngsters learn programming in BASIC or LOGO better than others. Clinical investigations of novice programmers suggest that this happens in part because different students bring different patterns of learning to the programming context. Many students disengage from the task whenever trouble occurs, neglect to track closely what their programs do by reading back the code as they write it, try to repair "buggy" programs by haphazardly tinkering with the code, or have difficulty breaking problems down into parts suitable for separate chunks of code. Such problems interfere with students making the best of their own learning capabilities: students often invent programming plans that go beyond what they have been taught directly. Instruction designed to foster better learning practices could help students acquire a repertoire of programming skills, perhaps with spinoffs having to do with "learning to learn." (29 references) (Author)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

85-13

ED305925

U.S. DEPARTMENT OF EDUCATION
Office of Educational Research and Improvement
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)

☒ This document has been reproduced as received from the person or organization originating it.

☐ Minor changes have been made to improve reproduction quality.

• Points of view or opinions stated in this document do not necessarily represent official OERI position or policy.

CONDITIONS OF LEARNING IN NOVICE PROGRAMMERS

Concept Paper

April 1985

Educational Technology Center
Harvard Graduate School of Education
337 Gutman Library Appian Way Cambridge MA02138

IR013794

Conditions of Learning in Novice Programmers

Writers: D. N. Perkins
Chris Hancock
Renee Hobbs
Fay Martin
Rebecca Simmons

Programming Group:

Betty Bjork
Chris Hancock
Renee Hobbs
Jack MacLeod
Fay Martin
David Perkins
Marie Salah
Nancy Samaria
Paul Shapiro
Rebecca Simmons
Tara Tuck
Martha Stone Wiske

Abstract

Under normal instructional circumstances, some youngsters learn programming in BASIC or LOGO much better than others. Clinical investigations of novice programmers suggest that this happens in part because different students bring different patterns of learning to the programming context. Many students disengage from the task whenever trouble occurs, neglect to track closely what their programs do by reading back the code as they write it, try to repair buggy programs by haphazardly tinkering with the code, or have difficulty breaking problems down into parts suitable for separate chunks of code. Such problems interfere with students making the best of their own learning capabilities: students often invent programming plans that go beyond what they have been taught directly. Instruction designed to foster better learning practices could help students to acquire a repertoire of programming skills, perhaps with spinoffs having to do with "learning to learn."

Conditions of Learning in Novice Programmers

Learning to program with some competence in languages like BASIC or LOGO poses a daunting challenge to many youngsters. Teachers of programming in primary and secondary schools frequently comment on the startingly different rates at which children progress. "Johnny can do anything, but Ralph just can't seem to get the hang of it." A series of clinical studies of young progammers conducted in association with the Educational Technology Center at the Harvard Graduate School of Education confirms this picture. The range of competence is indeed striking; the natural question is, "Why?"

Several factors may contribute to this phenomenon, including the limited instruction that students receive. In schools with only a few computers, students commonly use a machine for only a couple of hours a week. Furthermore, many teachers of programming in primary and secondary school are new to the enterprise, and are having to find their own way in a difficult, and sometimes unwelcome, new field. Finally, programming as a recent subject area has a relatively

undeveloped pedagogy, especially at the primary and secondary levels. In such circumstances, student achievement naturally sprawls across a wide range depending on such variables as intelligence, flair for computing specifically, presence of a computer in the home that allows more "hands on" learning, and so on.

Although these factors are important, they do not penetrate very deeply into the learning processes of young programmers. The issue is to identify what the most successful learners do that helps them to learn, in contrast to those who remain befuddled. Definitive answers to this question would require a massive program of research. However, our clinical studies allow some tentative conclusions about patterns in the behavior and attitudes of novice programmers that favor progress. Not only do these patterns help in understanding why some learn more than others but they also offer a guide to remaking instruction. The attitudes and behaviors that help students to learn might be directly encouraged by the teacher and the instructional program. We will return to this point.

A Note on Methodology

Although the aim of this paper is not to present a technical account of out research, we offer this brief note on our subjects and methods. Observations of two subject populations inform the perspective presented here. At one high school, we have been observing students from mixed-grade introductory BASIC classes that meet for one period per day. The period is devoted both to classroom discussion and to hands-on programming work. At an elementary school we have been

working with LOGO students from the fourth, fifth and sixth grades. These students learn LOGO less formally than the high-school BASIC students. There are two computers at the back of the classroom, and groups of two or three youngsters work at each computer while the rest of the class carries on with other subjects. At both of these sites, there was an initial stage in which we observed students as they worked in the classroom, after which we began to see students individually outside the classroom setting. In this latter stage we have observed approximately 30 BASIC students and 25 LOGO students.

The method of observation is a structured clinical one. An experimenter sits with a student and presents a programming problem. The experimenter observes as the student attempts to solve the problem, probing occasionally for explanations of the student's ideas and intentions. When the student encounters substantial difficulty, the experimenter intervenes with prompts designed both to help the student and to disclose the nature of the difficulty. Working within general guidelines, the experimenter tries to devise a revealing probe, responsive to the student's particular situation. The experimenter takes notes on what the student says and types and how the computer responds; in addition the session is audio tape-recorded. Later, a rough transcrioption is prepared, reflecting both the notes and the tape. The transcription is examined to track the thinking of the student and interpret the nature of the difficulties the student encountered.

For a simple example of a probe, suppose a student has already described his dilemma in a way that clearly calls for a FOR loop, but

the student has not retrieved that construct. The experimenter might ask, "Can you think of any instruction you know that might help with that?" If this yields nothing, the experimenter might go on to suggest, "Why don't you try a FOR loop?" Suppose the student then proceeds to apply a FOR loop correctly: in that case one has learned that the student has some ability to use the FOR loop construct but a problem with retrieving it on appropriate occasions. Through a number of such clinical interviews, a map of typical difficulties is emerging.

Initially, the probes were limited to high-level problem-solving advice, such as "What do you think the problem is right now," "If you don't see how to do thie whole thing, is there a part of it you can do," or "Can you read through the program and tell me exactly what it does?" The hypothesis was that many students had difficulty with programming in large part for lack of good high-level problem management skills. Gradually, however, a much more complex situation emerged, leading to the occasional use of more directive probes such as, "Why don't you try a FOR loop?" One discovery was this: far from being haphazard and unpatterned, many students' management of the task showed strong patterns that interfere both with the immediate programming problem and with learning. The present paper focusses on certain of these patterns and explores their import for the pedagogy of programming. Later articles will examine other results from these clinical studies.

The Possibility of Learning

It is important to appreciate the challenge that beginning programmers face. Seemingly straightforward problems can present appalling difficulties to the novice. For example, consider one problem sometimes used in our clinical research. The problem directs the student to write a BASIC program that will print a square of stars, where the number of stars on each side is to be specified by the user. For instance, if one enters a 5 in response to the program's prompt, the program will type back:

```
* * * * *
*       *
*       *
*       *
* * * * *
```

Although this problem will not baffle the experienced programmer, it challenges the beginner in a number of ways. For instance, many of the students we work with have never encountered a problem that requires a variable, much less an algebraic expression, for the upper limit of a FOR loop. Our square problem calls for upper limits of S and S-2, where S is the number of stars per side. Moreover, the students have never encountered a loop that iterates across a single line of output, as in the top and bottom rows, nor a circumstance requiring nested loops, as the middle rows of the square demand. They have rarely considered problems that must be tackled in parts, each segment of the program itself a unit that produces one section of an output that looks like a seamless whole.

We use this problem because it demands of many students that they employ the primitives they know in new ways: in other words, that

they invent. When this happens, they are "going beyond the information given", in Jerome Bruner's felicitous phrase. To put it another way, they are lifting themselves up by their own intellectual bootstraps. Such learning by discovery is entirely appropriate considering the open-ended character of programming problems. When we say that someone knows how to program, we are implying that that person can solve programming problems that are more than trivially different from problems he or she has encountered before.

Of course, it is also possible to prepare students more explicitly for the range of problems they will encounter. Skilled programming appears to depend on a repertoire of well-practiced schemas, as do other complex intellectual performances such as problem solving in mathematics and physics, and chess play (Chi, Feltovich, and Glaser, 1981; Chase and Simon, 1980; Larkin, McDermott, Simon and Simon, 1980; Schoenfeld and Herrman, 1982; Soloway & Ehrlich, 1984). Such schemata and accompanying reasoning tactics might be taught to students directly, rather than leaving students to figure them out for themselves as more typically happens. Soloway (Soloway, 1985; Soloway & Ehrlich, 1984) is investigating this approach in the teaching of Pascal, while Anderson and his colleagues are conducting a somewhat similar experiment, involving a computerized tutor, at the college level with the LISP programming language (Anderson & Reiser, 1985). There is every reason to think that these schema-based approaches to instruction will result in more efficient learning.

The point remains that, even without carefully designed instruction based on cognitive science and in-depth study of the particular

domain, some students learn. What might be called "bootstrap learning," where students go significantly beyond what they have been taught, does occur. Not only is this apparent from the occasional student with a flair for programming, but our clinical inquiries teach us that even students who are not doing that well overall can occasionally invent programming tactics for themselves. Consider these examples.

Sandra was working on some simpler problems in preparation for the square-of-stars task. First, she was asked to print a vertical row of ten stars. This posed no difficulty. Then she was asked to write a program that would accept a number from the keyboard and print that number of stars. Sandra puzzled for a moment and then said she thought she saw how to do it. Sandra read the number into a variable with an INPUT statement and used the variable as the upper limit of a loop. This would be completely unremarkable except for the fact that Sandra had never needed to employ a variable as the limit of a FOR loop in any previous work.

George, working on the square-of-stars problem itself, realized that he needed a loop to generate the middle lines of the square. Recognizing that there were S-2 lines, George asked, "Can I use something like S-2 in the loop?" Unremarkable again, except that George had never before used an expression in a FOR statement.

In LOGO, we often ask learners to write a program that will draw three identical rectangles in a stack, with some space between. Beverly had written code for the first of three rectangles. She was asked how she could use that code to create the subsequent two rectangles. Beverly employed the immediate mode to explore what would

happen if she executed the procedure she had just written with positioning moves to place the turtle in different locations on the screen. She asked the researcher, "Can I put this into a procedure?" -- referring to her work using three repetitions of the procedure she had already written. This is just what students are supposed to do in LOGO, of course. However, this student had never written programs involving embedded procedures, nor had she even been exposed to them as far as we could tell.

The students discussed here are just beginning to learn programming. Nonetheless, they display significant powers of invention. In our observations, many students can be seen inventing ways of tackling the particular problems they encounter. The difficulty is that a number of pitfalls in the process of programming can interfere with their making the best use of their problem-solving abilities to bootstrap themselves toward a reasonable level of competence. It seems likely that only the students who both invent solutions as illustrated here and avoid the pitfalls make steady progress. We now examine what some of these pitfalls are.

Stoppers and Movers

When novice programmers see fairly quickly how to proceed, naturally they do so. When, however, a clear course of action does not present itself, the young programmer faces a crucial branch point: what to do next? Try to break the problem down, look in the text for an idea, attempt something by trial and error?

Some students quite consistently adopt the simplest expedient and just stop. They appear to abandon all hope of solving the problem on their own. A dialogue like this is typical:

Student: I'm stuck.

Observer: What do you think the problem is?

Student: I don't know how to program it.

Observer: What ideas do you have?

Student: I don't know.

Lacking a ready answer to the difficulty, the student not only feels at a complete loss, but feels unwilling to explore the problem any further. We label learners who display this type of behavior "stoppers."

Other students consistently try one idea after another, writing or modifying their code and testing it, never stopping long enough to appear stuck. We call them "movers." Sometimes movers do well, making progress on a problem and carrying it through to a successful completion. Extreme movers, however, move too fast, trying to repair code in ways that, with a moment's reflection, clearly will not work. This approach sometimes leads students to abandon prematurely quite promising ideas because they don't work the first time. Moreover, the extreme mover often does not appear to draw any lessons from ideas that do not work. There is no sense of "homing in" on a solution. Indeed, the student may even go round in circles, retrying approaches that have already proven unworkable.

Stoppers and movers can be viewed as being at endpoints of a continuum based on the ratio of time spent thinking (or time spent sitting in front of a terminal and not typing) to time spent entering

and testing code. But this image of a continuum is in a way misleading. It suggests a distribution with most students in the middle while extreme stoppers and movers occupy the statistically rare tails. On the contrary, the descriptions of stoppers and movers are not caricatures of the norm. Extreme stoppers or movers are common.

Although, movers have their problems, at least they are in motion, with some hope of wresting a solution out of their venturing. Stoppers, on the other hand, have no chance of making progress, because they have given up. Stoppers represent the most obvious sign of the powerful affective factors we constantly see at work in novice programmers. Cindy, for example, had ground to a halt working on the square-of-stars problem. Asked if she had any ideas about the problem, she replied that she didn't. She was clearly waiting for the researcher to give her the answer. The researcher posed a question designed to relieve Cindy of immediate pressure, and also to dodge the request for help: "Suppose you were working on this problem in a room by yourself, with no-one to help. What would you try next?" Cindy replied in frustration that if she were by herself, and could do what she wanted, she wouldn't be working on a computer.

One would expect the affective element to have its greatest negative impact with novices. Such learners are most likely to feel unsure of what they are doing, harbor fears about handling the machine, and hold in doubt their ability to make the machine do what they want it to do. Computer work can be a challenging and stimulating experience, but also it can become a threat to self-esteem and one's standing with peers and teachers.

Another factor clearly affecting many students is their attitude toward making mistakes (Dweck and Licht, 1980). Some novices seem to take the inevitable occurrence of bugs in stride, while others become frustrated every time they encounter a problem. The former seem to recognize that mistakes are part of the process of programming, part of the challenge. They study their bugs and try to use the information they gain. The latter students appear to view bugs more as reflecting on the value of their performance. For them, programming mistakes can be devastating because the mistakes are so obvious -- they show up on the screen as incorrect output or in a program that will not run or gets stuck in the middle. We saw a striking example of this at one of our classroom observation sites. As a researcher walked by, one student hurriedly cleared his computer screen in order to conceal his program's erroneous output.

Such attitudes almost inevitably breed stoppers. Naturally, some stoppers become so disengaged that they learn very little. Surprisingly often, though, we have observed students who initially felt very reluctant to persevere in a programming task but, when encouraged, proved able to complete the task or at least a portion of it. For example, consider the case of Tom, a young student of BASIC. When the researcher first sat down with him, Tom was quick to point out that he had had no programming experience prior to this course, and that he didn't really know what he was doing. He said that the other students around him knew a lot more than he did, mentioning one student in particular who was working on a program for a game.

When Tom encountered difficulties he tended to leave the current problem and go on to the next, without making an effort to understand what was wrong. For instance, he encountered a bug when running a program that he had copied from the text. The program used array subscripts. When he ran it, he received the error message, "subscript out of range". He paused for a few seconds. Then, without a word, he began to look at the next problem. The researcher stopped him and asked what he thought the error message meant, to which Tom replied that he didn't know. When pressed for an answer, Tom thought for a little while, and then said that maybe the number in the parentheses need to be smaller. The researcher asked him to try this idea out at the computer. Tom tried a smaller subscript value, and the program ran successfully. Without further prompting, he went on to test various values until he had established the exact range of allowable subscripts.

Cases such as Tom's are cause for optimism: it may be that, with appropriate instruction and encouragement, students can learn not to give up so easily. The result may be enhanced learning and enjoyment for many students.

If stoppers illustrate the powerful influence of negative affect on students learning to program, certain movers in a different way show such an influence too. An extreme mover is also, in his own way, disengaging from the problem. While a successful mover is involved in the task, taking pleasure when things go right and acknowledging when they go wrong (one has to acknowledge a mistake before one can learn from it), the typical extreme mover seems emotionally more

distanced from the task. The keyboard and screen are a handy distraction, allowing the student to keep busy without pausing to think about the difficulties of the problem at hand. Instead of dealing with mistakes and the information they might yield, the extreme mover seeks to avoid them by moving on.

To summarize, many novice programmers exhibit to a greater or lesser extent the "stopper" syndrome, a kind of disengagement that reinforces itself by interfering with the student's development of greater competency. Stoppers, especially those who with encouragement prove able to do the problems posed, demonstrate by their conduct the powerful affective factors at work in the novel environment of computing. Although movers, by virtue of their motion, have at least some chance of solving a posed problem, impulsive, unreflective coding leaves some movers moving without getting anyplace.

Close Tracking of Code

A vital skill for any programmer is what we call "close tracking." Close tracking means reading written code to determine precisely what it does. Close tracking is useful for filtering out bugs before testing a program and also important for diagnosing and repairing bugs that appear when the program is run. Accurate close tracking is a mentally demanding activity. It requires understanding of the primitives of the language and the rules for flow of control. In addition, as the student proceeds through the code, the student must map its effects onto changes in what might be called a "status representation," specific to the problem. For instance, in LOGO

graphics problems, the turtle signals the status in large part; the student must read with precision how each piece of code alters the position and orientation of the turtle, and whether the pen is currently up or down. In BASIC, or in nongraphics LOGO problems, the status is best conceptualized as a matter of the values of variables and the appearance of accumulated output at a given point in time; the student must interpret accurately how each piece of code adds to the output or alters the value of a variable.

Although in principle close tracking is a mechanical procedure, in practice it often proves a source of difficulties. Students commonly neglect to do it when they need the information close tracking provides to untangle a problem. For example, in LOGO, Per was trying to make the design shown in figure 1, which builds on the three-rectangles problem.

Insert Figure 1 about here

He broke the problem down by creating one procedure to draw three rectangles, and another procedure to move the turtle diagonally from one block to the next. When he ran the positioning procedure, he could see that both the positioning angle and the length of his positioning move were wrong. Even though he could correctly diagnose the errors in the visual output, Per could not easily edit the program to repair his code. Repeatedly, as he tried to repair his program,

he would look at the whole program, saying "I think it's an angle problem." Then he would repair a line quickly, apparently with no thoughtful examination of the relationship between the code and the output. Close tracking would have led Per to a correct repair much sooner.

In a similar case, Fred was working on the square of stars problem in BASIC. He had written the following code:

```
10 n$ = "*"
20 input "how many stars per side";n
30 for x = 1 to n
40 print n$;
50 print n$
60 print n$;
70 next x
```

When he ran the program he got this output:

```
**
***
***
***
*
```

He repaired the program by changing line 40 to:

```
40 print n$,
```

so that the cursor would move to the next print zone. When he ran the program again he got this output:

```
*     *
**    *
**    *
**    *
*
```

Now the program was at least looking more like two sides of the square and he seemed to think that he was getting closer to the solution. Close tracking might have helped Fred to realize that his approach could not possibly work in the general case, because it allowed the

width of the figure to be determined by print zones instead of n, the desired width. But rather than track to see what the program was actually doing, Fred persisted in making many small repairs to lines 40, 50 and 60; he became stuck in a cycle of diagnosis and repair that could never get him closer to the correct programming solution. Through failure to track the program, students are more likely to follow a dead end path without even realizing they are doing so. Moreover, failure to track leaves them with few effective strategies for getting unstuck.

Several factors seem to contribute to students' failure to track. First, many students do not realize that tracking is an important programming strategy. In our research, students seldom tracked their programs without prompting. This phenomenon seems akin to the widespread tendency of students not to check their work in mathematical problem solving (cf. Polya, 1957; Davis, 1984; Schoenfeld, 1980). Failure to track may also result from a lack of confidence in one's abilities to predict or simulate the outcome of the program. Lack of confidence may result from the belief, articulated by some young LOGO students, that you can never be sure about what the computer will do. In other cases, lack of confidence may result from realizing that you do not completely understand how the language works.

Finally, for programs that have a graphic output, students may be discouraged from tracking by their visual perception of what is going on. In the cases of Fred and Per described above, both boys may have thought they understood what the program was doing because they could see a more or less accurate representation of the desired

product. In fact, their programs were much further away from being correct than the appearance of the output suggested.

Neglect of close tracking aside, when students do attempt to track what their programs are doing, they often fail. In its simplest form, this reflects an inadequate grasp of the primitives of the language, a problem that often arises. For example, Jane was making a square in LOGO using immediate mode. To create the square she successfully used the commands FD 90 RT 90. When asked to make a smaller square, she coded FD 30 RT 30. According to her mental model of making a square in LOGO she believed that the forward and turn inputs must be the same. Tracking the program would be unlikely to help her make an accurate repair until she acquires a correct concept of how RT 30 is different from RT 90.

For another example, in BASIC we have often observed confusion between the READ and INPUT statements for getting information into the program. Although the programs require inputs from the terminal, many students have coded READ statements, which do not get input from the terminal but rather from a data section that is coded directly into the program. When proofreading their code they seem to interpret such a statement loosely as the line that gets the numbers into the program. It has been suggested that in general people learn only the discriminations they have to in order to cope with the problem at hand (Davis, 1984). Some lapses in mathematical computation and reasoning, as well as the confusion noted here, follow from this principle. Students introduced to INPUT and later to READ (or vice versa) are likely to take their cue as to which to use from whichever

topic is current rather than from situational need; hence they do not at first learn how to differentiate sharply situations suited to one or the other.

In some respects tracking is similar to proofreading--reading back the code to make sure that you have recorded what you meant to write and that it is accurate in detail. As in proofreading and other contexts, errors get overlooked because one projects one's expectations on the stimulus, a marked feature of human perception generally (Neisser, 1967). We describe this type of failure to track accurately as projecting intentions on the code. For example, Tom was having trouble with a LOGO exercise because of the orientation of the turtle after drawing a rectangle. After the rectangle was drawn, the turtle was facing left. To complete his design, he needed the turtle to face north. Tom could not find his error and could not understand why the turtle was not facing up. He was prompted by the experimenter to track the program--to map his code onto the figure that was drawn. On successive attempts, he was simply unable to make the one-to-one correspondence. He would start off tracking precisely but, as he proceeded around the edges of the square, he would pay less and less attention to the code, extrapolating impressionistically from the previous sides.

We have discussed causes of difficulties with close tracking primarily in the context of programming. It is also worth noting that these influences may be aggravated in some students because of broad cognitive style traits that they bring to the programming task. Those students who naturally approach problems methodically

and reflectively may be better trackers than those who approach their work in a more trial-and-error, or impulsive, fashion (Kagan and Kogan, 1970).

In summary, we have observed how students' differential abilities and propensities to track their programs lead to more or less successful programming. Several factors may account for failure to track accurately: 1) motivational influences stemming from lack of understanding that tracking is important and lack of confidence in one's ability to track; 2) faulty understanding of how the programming language works; 3) projecting intentions onto the code so that one cannot objectively map the code as written onto the output; and 4) cognitive style differences.

Tinkering

Students often program by means of an approach we call tinkering -- they try to solve a programming problem by writing some code and then making small changes in the hopes of getting it to work. In some cases this strategy can be effective, while at other times it interferes with students' progress in solving programming problems.

For a positive example, Deborah, working on the three-rectangles problem in LOGO, wrote a long, unstructured string of code to draw the rectangles. When repairing the code, Deborah did not track systematically, but diagnosed her bug as "an angle problem." She tinkered with the code by making a series of small repairs and tests. Because this student was careful to correct the tinker if she found that an

earlier repair was incorrect, she was able to isolate the problem and complete her program.

On the other hand, tinkering often works out poorly. For example, Donald was also attempting the three-rectangles problem. He wrote a line of code to draw all three rectangles: REPEAT 3 [REC MOVE REC MOVE REC MOVE], where REC and MOVE were procedures he had defined earlier. The REC procedure drew a single rectangle and the MOVE procedure repositioned the turtle. The output drew nine rectangles instead of the desired three. Instead of re-thinking the problem, and questioning his understanding of how REPEAT works, he assumed that only some minor change was required, and so made repeated efforts to repair his code by rearranging the order of the procedures within the brackets.

The phenomenon of tinkering relates both to the theme of stoppers and movers and to the theme of close tracking. Tinkerers are by definition "movers", that is, they are not easily frustrated when a program fails the first time. In addition, tinkerers like to experiment with the code, so that they are not afraid of making changes. Finally, tinkerers hold the belief that the problem is solvable, and that they may be able, with their strategy, to find the solution. These important attitudes help a student to use tinkering as a problem-solving strategy.

At the same time, <u>effective</u> tinkering depends to an extent on close tracking. Those rare students who track very closely indeed are not tinkering at all; they know exactly what is going on. Students who do some tracking may succeed with a tinkering strategy, because even

though they do not know exactly what the problem is they have it somewhat localized. Students who track poorly or neglect to track may fall into a pattern of unrestrained and haphazard tinkering.

Students who tinker unsystematically often make their problems worse. For example, Adele, working on the three-rectangles problem in LOGO, tinkered with the code in a peculiar way. She made small changes in the code, and subsequently did not test to see the results. With no monitoring of the impact of her repairs, the program got worse and worse. Eventually, Adele wisely abandoned the procedure and started from scratch.

Tinkering is a particularly tempting trap when the program seems as though it is behaving nearly as desired. Fred's work on the square-of-stars problem, described in the previous section, is a case in point. Encouraged because his output resembled the target, Fred fell to tinkering with the program to try to achieve a complete match. Unfortunately, a completely different approach was needed. Fred was operating at the wrong level. In information processing terms, he was employing a hill-climbing strategy that encountered a local maximum: Fred could never make his output much better than it was by minor modifications. He needed to find a different hill to climb. As a generalization, tinkering may be helpful so long as the tinkerer is climbing the right hill -- has the right general approach -- and tinkers systematically, removing unsuccessful repairs so that the program does not become a tangle.

In summary, for novice programmers tinkering has both positive and negative features. On the positive side, it is a symptom of a

mover rather than a stopper: the tinkerer is engaged in the problem and has some hope of solving it. With sufficient tracking to localize the problem accurately and some systematicity to avoid compounding errors, tinkering may lead to a correct program. On the negative side, students often attempt to tinker without sufficient tracking, so that they have little grasp of why the program is behaving as it is. They assume that minor changes will help, when in fact the problem demands a change in approach. Finally, some students allow tinkers to accumulate untested or leave them in place even after they have failed, adding yet more tinkers until the program becomes virtually incomprehensible.

Breaking Problems Down

Breaking a programming task down into subproblems is a crucial skill for for the able programmer. There are few significant programming problems that can be solved by a single loop or one round of input, conditional branching, and output. Problems of any complexity call for partitioning the problem into parts, each one of those parts corresponding to a distinct segment of code and often to a separate syntactic unit in the program, such as a FOR loop, a REPEAT statement, or a subroutine call. The importance of breaking problems down is, of course, not limited to programming. It is one of the basic "weak strategies" of problem solving in general (Amarel, 1983; Newell and Simon, 1972; Lenat, 1983) and figures prominently in, for instance, heuristic approaches to mathematical problem solving (Polya, 1954, 1957; Schoenfeld, 1980, 1982; Wickelgren, 1974).

Ways of breaking problems into parts depend on the programming environment and the nature of the programming language, quite apart from the student's own intellectual skills. Students often program in LOGO by composing code in the immediate mode exclusively, so that their work more closely resembles a videogame-like maneuvering on a two-dimensional screen than programming. They then rewrite their code, often complete with false moves, backups and erasures, into one long procedure.

LOGO permits students to produce rather complex outputs without the use of high level programming concepts. Many turtle graphics problems are susceptible to a trivial kind of breaking down: the components are the segments of a target figure and the turtle need only trace them out, one by one. Neither BASIC problems nor non-graphics LOGO problems typically break down in this easy way. This property of LOGO is, of course, distinctly double-edged; it makes programming accessible even to rather young children, but does so through a simple linear pattern of thinking that does not generalize well to more sophisticated problems. Pea and Kurland (1984a,b) suggest that such programming is not cognitively demanding, but requires mainly stamina and determination.

When students do have to break down problems in nontrivial ways, they often falter. Some appear not to recognize the need to factor a programming problem into parts. Dorothy, for example, had solved a couple of easy problems in BASIC that demanded only a single organizational unit, a FOR loop with a preceding input statement. Then the experimenter posed the square of stars problem. After some moments

of puzzlement, Dorothy announced that she had no idea how to proceed with the problem. When the experimenter prompted her to break the problem down into parts, she showed some ability to do so; yet this strategy had not occurred to her at the outset.

When students do break a problem down, they often do not identify appropriate or workable chunks. They may factor the problem into subgoals that are not suited to the language or the task. Sometimes the natural perceptual characteristics of the programming goal lead students to formulate chunks that make the programming task more difficult or even impossible to solve.

For example, students addressing the square-of-stars problem in BASIC often break the problem down according to its visual components, without taking into account the constraints exercised by the primitives at their disposal. In a typical case, one student successfully coded the top row of stars, but then identified the left side of the square as the second subgoal, without recognizing that, because of the way the PRINT statement works in BASIC, both the left and right sides must be dealt with in the same chunk (unless one uses the LOCATE primitive, which can move the cursor to any position on the screen. LOCATE is not commonly taught to beginning BASIC students). This pitfall reflects the gestalt pull of the square of stars, with its natural decomposition into four sides. Ironically, in LOGO one proceeds with such a problem in just the way that BASIC students mistakenly lean towards -- one side at a time. In addition, the pitfall reflects an insufficient mental model of the way PRINT prints -- line by line -- or at least a failure to bring that knowledge

to bear in factoring the problem. Broadly speaking, students need to break problems down according to the sorts of operations the language affords rather than according to superficial features of the output.

Our observations of LOGO students suggest that students frequently approach a programming task without thinking out in advance the necessary components of the project. The strategy of planning as one codes may be feasible for expert programmers because they have at their disposal a well-developed repertoire of programming "plans" for different chunks of the programming task. But for novices with a scanty repertoire of programming plans, this often leads to an unworkable breakdown of the problem.

For example, Rodney, working on the three-rectangles problem, wrote a program named RECTANGLE which included, along with the rectangle itself, the positioning moves to place the cursor at the top of the screen. He encountered repeated difficulties using this procedure to make the other two rectangles. Because Rodney had decided somewhat arbitrarily, as he went along, where to chunk the problem, he ran into a series of troubling debugging problems that eventually led him to abandon the procedure and start afresh.

To summarize, several factors influence whether and how well students break problems down. The programming language affects the extent to which students can solve problems without the use of modular programming. Students do not always recognize that breaking problems down will aid in the solution of programming problems. Students often face trouble in attempting to break complex problems into

subgoals because their expectations interfere with planning: they use inappropriate analogies to drawing or other activities and fall under the influence of perceptual components of the programming goal. Lacking an accurate mental model of the language, its primitives, and the sorts of programming plans that can be built out of those primitives, students cannot easily discriminate inappropriate decompositions of problems. Finally, students face trouble in breaking problems down because they often try to deal with decomposition issues in the middle of coding, instead of planning deliberately in advance. Instructional intervention which encourages pre-planning might help.

Conditions of Learning versus Components of Expertise

The foregoing sections provide a broad-stroke description of some of the challenges novice programmers face. It was emphasized that beginners sometimes evince "bootstrap learning", inventing ways of handling particular programming problems, rather than applying only what they have learned in a narrow manner. At the same time, numerous factors interfere with their achieving such insights and making the best use of them to carry problems to completion. In particular, the affective and attitudinal profile of "stoppers" prevents them from exploring the problem situation. Neglect of close tracking or inability to track deprives students of crucial information about the behavior of their programs in progress. Tinkering at its worst mires learners deeper in mistaken approaches to a programming

problem. Difficulties in breaking a problem down bar programmers from proceeding with problems that involve much complexity.

These factors relate both to general problems of learning and to problems specific to programming. On the side of generality, it is clear that being a "stopper" inhibits progress in any domain, as does undirected tinkering. Attempting to break a problem down is a general problem-solving strategy. Close tracking reminds one of the difficulties many students encounter with any problem that requires high precision, as in arithmetic, for instance. At the same time, many aspects of the phenomena discussed seem specific to programming. The importance of close tracking relates to the activity of running a program and explaining its behavior, right or wrong, by examining the instructions one has written; in most other precision-demanding activities, such as arithmetic, one cannot "run" anything to test the adequacy of one's work. While breaking problems down is a general problem-solving strategy, good ways to break a problem down are of course conditioned by the nature of the programming environment; it was emphasized earlier, for instance, that BASIC and LOGO lead to somewhat different difficulties here.

The dual nature of the bahaviors discussed -- in some ways general and in some ways specific to programming -- is a positive feature since it indicates links with other subject matters, yet shows sensitivity to the particular circumstances of programming. Moreover, it appears likely that proper instruction might alleviate many of the problems identified. For instance, instruction could simply emphasize the value of certain practices: being a mover

rather than a stopper, tracking closely, seeking ways to break problems down. This follows from our observation that, quite apart from ability, simple neglect of such strategies is often a difficulty. In addition, instruction might enable students to carry out key activities much better than they typically do. For example, teaching a mental model of the machine in relation to the particular programming language, treating it as a "glass box" instead of a black one (DuBoulay, O'Shea and Monk, 1981), should help students both with close tracking and with decomposing problems in ways fitting the language's capacities. Research by Mayer (1981) testifies that teaching such a model can improve programming performance. Students could be taught explicitly the pitfalls of tinkering, and could be encouraged in such practices as removing failed repairs and considering, if several tinkers fail, whether a completely different approach needs to be taken.

These implications for teaching complement in an interesting way contemporary approaches to programming instruction based on studies of expertise. As noted earlier, expert performance in any domain appears to depend on a sizable repertoire of schemata that skilled individuals marshal in dealing with particular tasks. Soloway (1985), developing this approach for teaching Pascal, discusses such "programming plans" as a counter variable, where a statement like N=N+1 serves to increment a variable and thereby keep count of items or events, or an accumulator variable, where a statement like A=A+MORE serves to keep a running total of numbers. As these examples suggest, such programming plans go considerably beyond the primitives of the language in question. They constitute purposeful units of code that serve

particular functions. A programming problem can be analyzed in terms of the functions needed and appropriate programming plans retrieved from one's repertoire to implement a program. Soloway's approach to instruction involves teaching such plans and their use directly.

Anderson and his colleagues are pursuing a related line of research aimed at developing a computer tutor for the programming language Lisp. They analyze expertise in Lisp programming as a production system (Newell & Simon, 1972) in which, roughly speaking, particular problem characteristics trigger the installation of particular programming plans (Anderson & Reiser, 1985). The computer tutor incorporates certain principles of tutoring, based on Anderson's ACT theory and the general literature on learning, to equip students with a repertoire of productions that enables them to program.

Both Soloway's and Anderson's approaches seek to supply students rather directly with the schemata that, in normal instruction, a few students work out for themselves. The question remains why some students do and others do not. We have argued here that certain broad attitudes and conducts interfere with students discovering their own programming plans, behaviors such as stopping, neglect of close tracking, casual tinkering, and neglect of or systematic errors in breaking problems down. If instruction were to address these problems, more students might build their own schematic repertoires without elaborately choreographed instruction in the component schemata of expertise.

Each approach has its merits. The "expertise" approach almost certainly promises faster learning and greater competence for the

sorts of programming tasks being addressed. It also tends to boost the performance of weak learners relative to strong ones, so that the former are not left as far behind as in conventional instruction. On the other hand, the "expertise" approach, as a general approach to instruction in any subject matter, requires an elaborate analysis of every skill so treated, something not likely to be carried out for all the many skills people need to acquire and also more readily carried out for rather formal tasks like programming or mathematical problem solving than for tasks such as literary criticism or philosophical analysis. Moreover, instruction closely targeted on a particular performance tends not to transfer as well to related performances (Salomon and Perkins, 1984; with reference to programming, see Mayer,1981).

A "bootstrap learning" approach based on the principles identified here and others like them might yield stronger transfer to other programming languages or nonprogramming tasks and might equip the learner to deal better with other learning situations where a schematic repertoire is not being provided explicitly. Certainly Seymour Papert's ambitions for LOGO (Papert,1980) reflect such a hope. Recent findings on the efficacy of LOGO suggest that students typically do not develop much mastery of the language nor the associated metacognitive skills (Pea and Kurland, 1894a,b). However, some of the factors behind this shortfall have been discussed here. Instruction designed to foster bootstrap learning but not providing an explicit schematic repertoire might produce competent and flexible programmers, and might yield the broad cognitive ripple effects some advocates

of programming instruction have hoped for (cf. the positive results on LOGO reported by Clements and Gullo, 1984).

On the other hand, any instructional approach that depends on bootstrap learning probably will prepare students somewhat less efficiently in the content directly addressed. Moreover, such an approach should not be thought of as easier to mount than an expertise approach. To be sure, one need not build an elaborate model of the component schemata. However, the instruction must be carefully planned to ameliorate the barriers to bootstrap learning identified here, and, no doubt, others, and in general to mediate strongly the learning process. Without careful mediation, students are unlikely to take the learning opportunities often enough for substantial gains to occur (Feuerstein, 1980; Delclos, Littlefield, & Bransford, 1985).

In summary, the relatively undeveloped pedagogy of computer programming instruction falls short on two fronts. It does not teach directly the sorts of schemata a skilled programmer needs. Moreover, it does not guide students in ways that foster bootstrap learning of those schemata. In consequence, youngsters vary widely in their progress, succeeding only to the extent that they happen to bring with them characteristics that make them good bootstrap learners in the programming context. Advances in either direction should do better by young students of programming.

REFERENCES

Amarel, S. (1983). Problems of representation in heuristic problem solving: Related issues in the development of expert systems. In R. Groner, M.Groner, & W.F. Bischof (Eds.), Methods of heuristics. Hillsdale, NJ: Erlbaum.

Anderson, J.R., & Reiser, B.J. (1985). The LISP tutor. Byte, 10(4), 159-175.

Chase, W.C., & Simon, H.A. (1973). Perception in chess. Cognitive Psychology, 4, 55-81.

Chi, M., Feltovich, P., & Glaser, R. Categorization and representation of physics problems by experts and novices. Cognitive Science, 5, 121-152.

Clement, D.H., & Gullo, D.F. (1984). Effects of computer programming on young children's cognition. Journal of Educational Psychology, 76(6), 1051-1058.

Davis, R.B. (1984). Learning mathematics: The cognitive science approach to mathematics education. Norwood, NJ: Ablex.

Delclos, V.R., Littlefield, J., & Bransford, J.D. (1985). Teaching thinking through LOGO: The importance of method. Roeper Review, 7(3), 153-156.

DuBoulay, B., O'Shea, T., & Monk, J. (1981). The black box inside the glass box: presenting computing concepts to novices. International Journal of Man-Machine Studies, 14, 237-249.

Dweck, C.S., & Licht, B.G. (1980). Learned helplessness and intellectual achievement. In J. Garbar & M. Seligman (Eds.), Human Helplessness. New York: Academic Press.

Feuerstein, Reuven. Instrumental enrichment: An intervention program for cognitive modifiability. Baltimore: University Park Press, 1980.

Hawkins, J. (1984). Computers and girls: Rethinking the issues. Bank Street Technical Report no. 24 .. [details to come]

Kagan, J., & Kogan, N. (1970) Individuality and cognitive performance. In P. Mussen (Ed.), Carmichael's manual of child psychology (vol 1). New York: Wiley.

Larkin, J.H., McDermott, J., Simon, D.P., & Simon, H.A. Modes of competence in solving physics problems. Cognitive Science, 4, 317-345.

Lenat, D.B. (1983). Toward a theory of heuristics. In R. Groner, M. Groner, & W.F. Bischof (Eds.), Methods of heuristics. Hillsdale, NJ: Erlbaum.

Mayer, R.E. (1981). The psychology of how novices learn computer programming. Computing Surveys, 13(1), 121-141.

Neisser, U. (1967). Cognitive psychology. New York: Appleton-Century-Grafts

Newell, A., & Simon, H. (1972). Human problem solving. Englewood Cliffs, NJ: Prentice-Hall, Inc.

Papert, S. (1980) Mindstorms: Children, computers, and powerful ideas. New York: Basic Books.

Pea, R.D., & Kurland, M.D. (1984a, August). Logo programming and the development of planning skills. Paper presented at the Conference on Thinking, Harvard Graduate School of Education, Cambridge, MA.

Pea, R.D., & Kurland, M.D. (1984b). On the cognitive effects of learning computer programming. New Ideas in Psychology, 2(2), 137-168.

Polya, G. (1954). Mathematics and plausible reasoning (2 vols.). Princeton, NJ: Princeton University Press.

Polya, G. (1957). How to solve it: A new aspect of mathematical method (2nd ed.). Garden City, NY: Doubleday.

Salomon, G. & Perkins, D.N. (1984, August). Rocky roads to transfer: rethinking mechanisms of a neglected phenomenon. Paper presented at the Conference on Thinking, Harvard Graduate School of Education, Cambridge, MA.

Schoenfeld, A.H. (1980). Teaching problem-solving skills. American Mathematical Monthly, 87(10), 794-805.

Schoenfeld, A.H. (1982). Measures of problem-solving performance and of problem-solving instruction. Journal for Research in Mathematics Education, 1982, 13(1), 31-49.

Schoenfeld, A.H., & Herrman, D.J. (1982). Problem perception and knowledge structure in expert and novice mathematical problem solvers. Journal of Experimental Psychology: Learning, Memory, and Cognition, 8, 484-494.

Soloway, E. (1985). Presentation at Harvard University, February 12.

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. IEEE Transactions on Software Engineering, SE-10(5), 595-609.

Wickelgren, W.A. (1974). How to solve problems: Elements of a theory of problems and problem solving. San Francisco: W.H. Freeman and Co., 1974.

Figure 1: